An Optimization Framework for Dynamic, Distributed Real-Time Systems*

Klaus Ecker†, David Juedes, Lonnie Welch, David Chelberg, Carl Bruggeman, Frank Drews, David Fleeman, David Parrott‡, and Barbara Pfarr‡

Abstract. This paper presents a model that is useful for developing resource allocation algorithms for distributed real-time systems that operate in dynamic environments. Interesting aspects of the model include dynamic environments, utility and service levels, which provide a means for graceful degradation in resource-constrained situations and support optimization of the allocation of resources. The paper also provides an allocation algorithm that illustrates how to use the model for producing feasible, optimal resource allocations.

Keywords: Resource Management, Application Adaptation, QoS, Utility, Real-time

1 Introduction

The use of distributed computing technology in real-time systems is increasing rapidly. For example, an important aspect of the NASA Earth Science vision is its sensor-web, an integrated, autonomous constellation of earth observing satellites that monitor the condition of the planet through a vast array of instruments. While this concept offers numerous benefits, including cost reduction and greater flexibility, its full potential cannot be realized with today's information system technology. Common real-time engineering approaches use "worst-case" execution times (WCETs) to characterize task workloads *a priori* (e.g., see [15, 16]) and allocate computing and network resources to processes at design time. These approaches unnecessarily limit the functions that can be performed by spacecraft and limit the options that are available for handling unanticipated science events and anomalies, such as overloading of system resources. These limitations can mean loss of scientific data and missed opportunities for observing important terrestrial events. As noted in [7, 13, 19, 20], characterizing workloads of real-time systems using *a priori* worst-case execution times can lead to poor resource utilization, and is inappropriate for applications that must execute in highly dynamic environments.

Adaptive resource management (ARM) middleware (software that resides between the computer's operating system and the computer's applications) can address this problem by dynamically reconfiguring the way in which computing and network resources are allocated to

---

* This work was funded in part by the NASA Earth Science Technology Office Advanced Information Systems Technology Program; by the NASA Computing, Information and Communications Technology Program; and by the DARPA Program Composition for Embedded Systems Initiative.

† Department of Computer Science, Technical University of Clausthal, 38678 Claudthal-Zellerfeld, Germany (ecker@informatik.tu-clausthal.de)

‡ Center for Intelligent, Distributed and Dependable Systems, School of Electrical Engineering and Computer Science, Ohio University, Athens, Ohio - 45701 (juedes | welch | chelberg | bruggema | drews | david.fleeman | david.parrott @ohio.edu)

‡ Real-Time Software Engineering Branch, NASA Goddard Space Flight Center, Baltimore, Maryland - 20771 (barbara.pfarr@gsfc.nasa.gov)

processes. In [15], we examined a command and control system in use by NASA, and explored how the components of that system could be distributed across multiple processors in such a way that the system remained as robust as before, and at least as capable of meeting its real-time processing requirements. We found that many benefits would be realized by treating related systems as one unified system that shares a *dynamically allocated pool of resources*. In [16], we explored the possibilities of adaptive resource management for onboard satellite systems. Satellites are now sophisticated enough to have multiple onboard processors, yet they generally have processes statically assigned to each processor. Little, if any, provision to dynamically redistribute the processing load is provided. Onboard instruments are capable of collecting far more data than can be downloaded to the Earth, thus requiring idle times between downloads. Although download times are known *a priori*, failed downloads can cause the buffer on the satellite to overflow. To handle this situation, ARM middleware autonomously determines the following: the allocation of resources to tasks, the fidelity of data processing algorithms (such as a cloud cover detection algorithm [1]), the compression type to use on data, when and what to download, whether data should be discarded, and the interval for gathering telemetry data from various onboard subsystems. Decisions are made based on a system-level benefit optimization that takes into account observation schedules, future and current download opportunities, satellite health, user-defined benefit functions, and system resource utilization.

To allow future research efforts in ARM to build upon the foundation that we have established, this paper presents our model of dynamic, distributed real-time systems. It also provides an algorithm that shows how to employ the model to perform adaptive resource management. In [22, 23] we presented static models for resource allocation of real-time systems, and in [24, 25] we presented dynamic models. Applications of our dynamic models [26, 27, 28] showed their effectiveness for adaptive resource management. However, our previous approaches lacked the information needed to gracefully degrade performance in overload situations, did not support feasibility analysis or allocation optimization, did not consider security aspects, and did not include network hardware. This paper removes those shortcomings by extending the model to incorporate knowledge of application profiles, network hardware, utility, and service level constructs.

The remainder of the paper is organized as follows. Section 2 presents the model. In Section 2.1, the model of the hardware resources is presented. Section 2.2 describes the model of the software system, which consists of subsystems, end-to-end paths, and applications (tasks). Section 3 shows how to use the model to check global allocation constraints and to perform global allocation optimization. A detailed framework for developing allocation algorithms based on the model is provided in Section 4. An overview of related research is provided in Section 5.

2 Mathematical Modeling

A dynamic real-time system is composed of a variety of software components that function at various levels of abstraction, as well as a variety of physical (hardware) components that govern the real-time performance of the system.

2.1 Hardware components

The physical components of a real-time system can be described by a set of computational resources and network resources. The computational resources are a set of host computers $H =$

$\{h_1,\ldots,h_m\}$. The properties of each host $h \in H$ are specified by a set of attributes, among them the more important ones are the identifier *name*(h), the size of the local read only memory *mem*(h), a numerical value *sec*(h) that specifies the current security level of h, speed factors *int_spec*(h) and *float_spec*(h) for the integer and floating point SPEC rates respectively, and overhead time $o(h)$ for send and receive operations. Computational resources are off-the-shelf general purpose machines running multitasking operating systems.

The network structure may be formalized as a directed graph $N = (H,L)$ where L is the set of physical (undirected or directed) links between host nodes. Each link $l \in L$ has a fixed *bandwidth bandwidth(l) and* operates in a mode *op_mode(l) which is either half duplex* or *full duplex*. We describe the connections between hosts by a function *link* : $H \times H \rightarrow L$. It is furthermore assumed that pairs of hosts h and h' are connected by a fixed communication path described by a function *route*: $H \times H \rightarrow P(T)$ where $P(T)$ is the set of all simple paths in T describing the basic routing information. Associated with each pair (h_1,h_2) of hosts is a *propagation delay* $p\_delay(h_1,h_2)$ measured in either packets per second or bits per second. An additional queuing delay may be considered in case of heavy communication load.

It is generally *assumed* that the set of resources and network topology are fixed.

2.2 Software components

While we assume that the hardware resources are fixed, the parameters that effect the performance of the software components may change dynamically. Nevertheless, we assume that the operating conditions and parameters of the software components are constant at least for some time interval.

To software components of a dynamic real-time system can be decomposed in several abstraction levels: the system, consisting of several sub-systems, each being a set of paths of application software (see Figure 1).

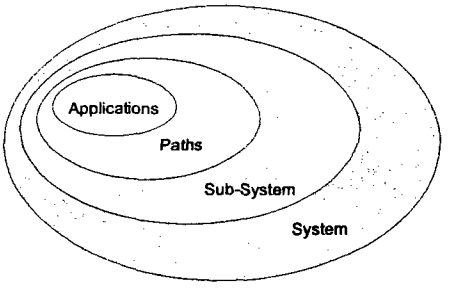

Figure 1: System hierarchy

2.2.1 The System The highest level of abstraction represents the *system*. A system $S = \{ SS_1, \ldots, SS_{m_1} \}$ is considered as a collection of *sub-systems*. There are no specific attributes associated with a system. It simply represents the entire set of sub-systems that are currently being executed on a single system.

2.2.2 Sub-Systems The next level of abstraction is that of *sub-systems*. A sub-system represents some part of the system that can be separated semantically from the total system. A sub-system

$SS = \{P_1, \ldots, P_{m_2}\}$ is simply a collection (set) of paths, along with a priority *prio*(*SS*) and a security level *sec*(*SS*). The priority is user-defined and determines the perceived priority of the given collection of paths. The sub-system priority and security level are inherited by the paths and applications in the sub-system.

2.2.3 Paths The next lower level of abstraction in a real-time system is the notion of a *path*. A path P_i consists of a set of applications $A_i \subseteq A$ and a precedence relation $\prec_i$. The precedence relation provides information concerning the execution order of the applications in a path, as well as their communication characteristics. We will assume that the transitive closure of the precedence relation $\prec_i$ defines unique largest and smallest elements in A_i. Different paths may share the same application, as shown in Figure 2.

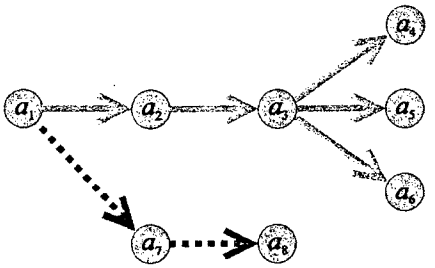

Figure 2: Example for overlapping paths

There are two basic types of paths: *periodic* paths and *event-driven* paths. Each periodic path P_i has a given period π_i. Modeling a periodic path implies that the path has to be executed exactly once in each period. Each event driven path P_i has a maximum event rate r_i, which is generally not known, and a deadline d_i. It is assumed that the deadlines are hard in the sense that it is not allowed to complete a path later than the deadline. In this paper, we model event driven paths as periodic paths where the period is the inverse of the event rate, $\pi_i = 1/r_i$. The reason is that, choosing $1/r_i$ as the period, covers the worst case scenario: if the paths can be scheduled feasibly with maximum event rates, then we are sure to have a feasible situation in case of smaller event rates. In each period there is a deadline that is d_i time units after the start of the period.

There are two more attributes: Each path P_i has a dynamic workload $w(P_i)$ that is essentially defined by the amount of input data for P_i, and a priority that is inherited from the sub-system P_i belongs to: $prio(P_i) := prio(SS)$.

As for the notation, the paths' workloads and maximum event rates are collected in vectors, the workload vector $\vec{w}$ and event rate vector $\vec{r}$, respectively.

2.2.4 Applications At the lowest level of abstraction, the software components of a real-time system consist of a set of *applications* $A = \{a_1, \ldots, a_n\}$. Each application a has some workload w_a. For simplicity we assume that applications in a path inherit the workload of the path: application a of path P_i has workload $w_a = w(P_i)$. Thus, overlapping paths (i.e. paths that have common applications as in Figure 2) have equal workloads.

One of the main objectives is to find an optimal allocation of the applications to host computers. Such an allocation, formally described by a function $host : A \rightarrow H$, has to fulfill runtime conditions and memory limitations on the hosts . Both, execution time and memory usage of an application depend not only on its workload and service level parameters, but also on the machine on which it is being executed.

We assume that there exists a set of n global *service levels* [6] $S = \{s_a \mid a \in A\}$ (one for each application), each of which may be set to an arbitrary value of $\mathbb{R}$. This (potentially multidimensional) parameter affects the level of service to the user, and therefore affects the overall utility of the system. Service level setting is defined for each application separately. This parameter also affects the running time of the application.

For each application $a \in A$, each host $h \in H$, each workload $w \in \mathbb{N}$ and each service level $s \in \mathbb{R}$, we define $r_{a,h}(w_a, s_a)$ as the processing time, i.e., the amount of time that a response requires when an application a is executed on host h with workload w_a and service level parameter s_a [11]. Similarly, $m_{a,h}(w_a, s_a)$ is the amount of memory used by application a in the same setting. In addition, to avoid non-eligible or security violating allocations, we make the following assumptions concerning $r_{a,h}$ and $m_{a,h}$:

(i) $r_{a,h}(w_a, s_a) = \infty$ and $m_{a,h}(w_a, s_a) = \infty$ if application a cannot be executed on h. This may occur if h is not an eligible host for a, or if there would be a security violation if a were to be executed on h.

(ii) Both $r_{a,h}(w_a, s_a)$ and $m_{a,h}(w_a, s_a)$ are assumed to be monotonically non-decreasing in w_a and s_a, i.e.,

if $w_a \leq w_{a'}$ and $s_a \leq s_{a'}$ (component-wise),

then $r_{a,h}(w_a, s_a) \leq r_{a,h}(w_{a'}, s_{a'})$ and $m_{a,h}(w_a, s_a) \leq m_{a,h}(w_{a'}, s_{a'})$.

If applications of a path are allocated to different hosts, data transmission between the hosts will be required. If $a \prec_i a'$, it is assumed that application a and application a' communicate via interprocess communication in the local area network. The amount of communication in a path depends on the workload of the path. Given a workload w_a and a setting of the service level s_a , application a sends $c_{a,a'}(w_a, s_a)$ bits of information to a'. We assume that $c_{a,a'}(w_a, s_a)$ is a monotonically non-decreasing function of the workload of a.

For each $a \in P_i$, a *priority* may be associated by defining $p_a := prio(SS)$ where SS is the *uniquely defined* sub-system that holds a path with application a. Priorities are useful to achieve certain overall system objectives.

3 The Resource Manager

The resource manager (RM) is responsible for the correct operation of the whole system. As input, it is given the static characteristics of both the hardware system and the software systems. The resource manager can not modify these properties. However, the resource manager is responsible for making all resource allocation decisions and has the ability to modify certain

performance parameters such as service levels. In this section we consider the constraints that must be satisfied and the optimizations that a resource manager can perform.

3.1 Global Allocation Constraints

In all situations the resource manager must provide an allocation that meets the constraints of the system. The proposed framework supports three constraints. First, the resource manager must ensure that each application is assigned to a valid host, one that is capable of executing the application. Second, the security level of each application should not be larger than the security level of both the assigned host and any communication links used in the current path. Third, the amount of time needed by any path to complete execution must not exceed the required deadline. The minimum responsibility of the resource manager is to choose an allocation of applications to hosts such that these three constraints are satisfied at a given setting for service levels, workloads, and arrival rates. A *feasible* solution is the specification of a function *host*: $A \rightarrow H$ that satisfies all the allocation constraints.

3.2 Global Allocation Optimizations

In addition to constraint-satisfaction, a resource manager should have the ability to perform various allocation optimizations. The objective is to find an allocation and setting of unknown performance parameter values such that all applications can be scheduled feasibly and the overall utility is maximized. The proposed model supports three performance parameters: maximum workload, maximum event rate, and service level. The workload and maximum event rate of an application are generally unknown. For this reason, the resource manager attempts to maximize the arrival rate and workload that can be handled by a particular allocation according to some heuristic. In addition the service level of an application is a knob that the resource manager can use to adjust both the resource usage and the overall utility.

The *overall utility* of a system can be determined from the maximum workloads, maximum event rates and service levels that are computed by the optimization heuristic. We formalize the overall utility as a function $U(S) = \bar{U}(\vec{s}, \vec{w}, \vec{r})$. Depending on the given characteristics of the system, there are many ways to specify such a function. An example system requiring fair distribution of resources is Dynbench [need reference], a shipboard missile detection and guidance system. The following *product* utility functions are able to handle such scenarios:

$$U_1(S) = \bar{U}(\vec{s}, \vec{w}, \vec{r}) = U(\vec{s}) \cdot \min_{a \in A}\{w_a\} \cdot \min_{a \in A}\{r_a\}$$

$$U_2(S) = \bar{U}(\vec{s}, \vec{w}, \vec{r}) = U(\vec{s}) \cdot \min_{a \in A}\left\{\frac{w_a + 1}{p_a}\right\} \cdot \min_{a \in A}\left\{\frac{r_a + 1}{p_a}\right\},$$

These functions can be used to prevent the resource starvation of lower priority applications. A *weighted sum* utility function does not prevent resource starvation, but allows higher priority Applications to obtain as many resources as needed for critical operation. These are example weighted sum utility functions:

$$U_3(S) = \bar{U}(\vec{s}, \vec{w}, \vec{r}) = c_1 \cdot U(\vec{s}) + c_2 \cdot \min_{a \in A}\{w_a\} + c_3 \cdot \min_{a \in A}\{r_a\}$$

$$U_4(S) = \bar{U}(\vec{s}, \vec{w}, \vec{r}) = c_1 \cdot U(\vec{s}) + c_2 \cdot \min_{a \in A}\left\{\frac{w_a + 1}{p_a}\right\} + c_3 \cdot \min_{a \in A}\left\{\frac{r_a + 1}{p_a}\right\}.$$

Some systems may require more complex utility functions. For example, we could combine the functions defined above in the following way:

$$U_\alpha(S) = \alpha \cdot U_2(S) + (1-\alpha) \cdot U_4(S),$$

where $\alpha \in [0,1]$ can be used as a control parameter to mix the strategies explained above.

The considerations and algorithmic approach presented in the remainder of this paper require monotonicity as an important property of the overall optimization function $U(S) = \bar{U}(\vec{s}, \vec{w}, \vec{r})$, which can be described as follows:

$$\vec{s} \le \vec{s}' \Rightarrow U(\vec{s}) \le U(\vec{s}') \qquad [\vec{s} \le \vec{s}' \text{ means component-wise}],$$

$$\vec{w} \le \vec{w}' \Rightarrow min(\vec{w}) \le min(\vec{w}') \qquad \left[\text{or } \min_{a \in A}\left\{\frac{w_a + 1}{p_a}\right\} \le \min_{a \in A}\left\{\frac{w'_a + 1}{p'_a}\right\}\right],$$

$$\vec{r} \le \vec{r}' \Rightarrow min(\vec{r}) \le min(\vec{r}') \qquad \left[\text{or } \min_{a \in A}\left\{\frac{r_a + 1}{p_a}\right\} \le \min_{a \in A}\left\{\frac{r'_a + 1}{p'_a}\right\}\right].$$

3.3 Considerations on Constraints and Optimizations

The following considerations are helpful in understanding our algorithmic approach defined in the next section. Assume for simplicity that instead of $\vec{s}$, $\vec{w}$, $\vec{r}$ there are only two system parameters, p_1 and p_2. Each may attain integer values ≥ 0. So the question is for which pairs (p_1, p_2) the system behaves correctly and utility $\bar{U}$ has a maximum. From the monotonicity assumption we conclude that we need only to look for pairs (p_1, p_2) that are maximal: (p_1, p_2) is maximal if each pair (p_1', p_2') $\ne$ (p_1, p_2) with $p_1' \ge p_1$ and $p_2' \ge p_2$, does not allow a feasible solution. Feasibility is checked by means of a heuristic algorithm such as threshold accepting, or simulated annealing, by directly finding an allocation of applications to host.

Maximal pairs can be determined by a systematic search: First one would find upper limits separately for p_1 and p_2 , while keeping the other value at minimum. Let p_1^{limit} and p_2^{limit} the respective maximum values. This can be done by a doubling strategy, by starting with 1 for p_1 resp. p_2. With known values p_1^{limit} and p_2^{limit}, an off-line algorithm could determine maximum pairs (p_1, p_2) with $0 \le p_1 \le p_1^{limit}$ and $0 \le p_2 \le p_2^{limit}$. Since we assume non-negative integer parameters, the number of pairs to check is limited by $(p_1^{limit} + 1)(p_2^{limit} + 1)$. Figure 3 illustrates the maximum parameter pairs (black dots). The pairs lying below and left of each maximum parameter pair allow feasible solutions.

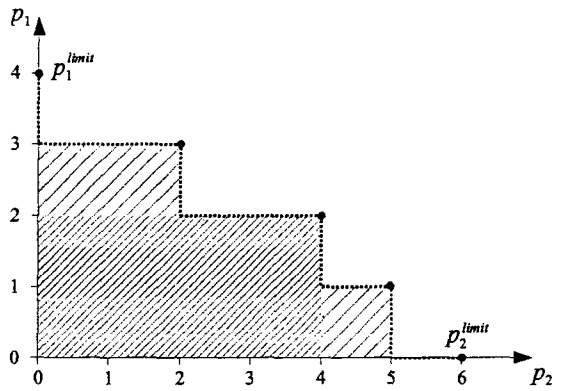

Figure 3: Determining the Boundary of the Feasible Region.

The generalization to the general parameter set $\vec{s}$, $\vec{w}$, $\vec{r}$ is straightforward. Knowing the parameter area with feasible solutions is useful for on-line algorithms; if the running system requires certain parameter settings, the feasibility of the settings can be checked easily.

4 A Framework for Allocation Algorithms

In this chapter a framework for allocation algorithms is presented with the objective to maximize overall utility. The utility of an allocation is a function of the service levels, calculated maximum workloads and calculated maximum event rates. The utility function does not depend on the particular structure of a solution, but assumes that the schedule is feasible.

Before discussing allocation algorithms, we must explore the differences between off-line and on-line algorithms. *Off-line algorithms* are performed before a system has been started, and thus are not limited by tight time constraints. For this reason, these algorithms may be brute force algorithms that are capable of finding optimal allocations and performance parameter settings. *On-line algorithms* on the other hand are executed simultaneously with the dynamic systems for which they are responsible for allocating resources. These types of algorithms operate under strict timing constraints and are typically used for making fast, intelligent reallocation decisions.

The framework proposed in this section is decomposed into several modules. An off-line algorithm could take advantage of all the functionality provided by these modules. In contrast an on-line algorithm may require the use of only a subset of the modules presented due to strict timing requirements. For this reason, we will look at each of the modules in the context of an off-line algorithm. The structure of such an algorithm is shown in Figure 4.

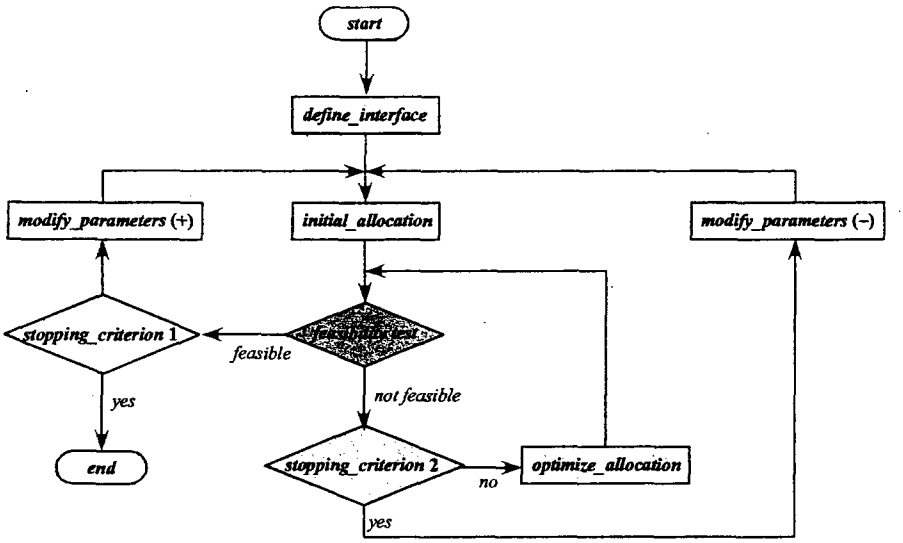

Figure 4: Structure and Modules of an Off-line Allocation Algorithm.

Initially, the *define_interface* module uses the hardware and software specifications to determine the initial settings for the triplet $(\vec{s}, \vec{w}, \vec{r})$ and the corresponding overall utility $\bar{U}(\vec{s}, \vec{w}, \vec{r})$. The *initial_allocation* module constructs an allocation of applications to hosts subject to the conditions of the triplet $(\vec{s}, \vec{w}, \vec{r})$. The *feasibility_test* module determines whether the allocation is feasible. If the allocation is feasible and *stopping_criterion1* has not been satisfied, then the *modify_parameters(+)* module increases the performance parameter settings resulting in a new setting for $(\vec{s}, \vec{w}, \vec{r})$ thus increasing the overall utility. However, if the allocation found was not feasible and *stopping_criterion2* has not been satisfied, then the *optimize_allocation* module modifies the allocation subject to the triplet $(\vec{s}, \vec{w}, \vec{r})$ by using optimization procedures such as general local search procedures, and greedy heuristics. This step continues until a feasible allocation is found or *stopping_criterion2* is satisfied. If *stopping_criterion2* is satisfied and the allocation is still not feasible, then the *modify_parameters(-)* module decreases the performance parameter settings resulting in a new setting for $(\vec{s}, \vec{w}, \vec{r})$ causing the overall utility to decrease. After either the *modify_paramteres(+)* or *modify_paramters(-)* has been executed, the algorithm reenters the *initial_allocation* module and the process continues. We will now look at each of these modules in more detail.

4.1 Module define_interface

The module *define_interface* provides interfaces between the resource manager and the allocation algorithm and provides the data structures to store the needed information for the operation of the allocation algorithm. The resource manager provides the module with the static characteristics of both the hardware and software systems as described in section 2. The module uses this information to produce initial settings for the unknown performance parameters and the service level of each application. These initial settings are represented by the triplet $(\vec{s}, \vec{w}, \vec{r})$.

The module returns this triplet and the corresponding initial overall utility $\bar{U}(\vec{s}, \vec{w}, \vec{r})$. Figure 5 provides more detail about this module. For example, the *latency* function in Figure 5 represents the actual amount of time the task will take to complete processing due to the resource needs of

the application, the resource characteristics, and the contention for the needed resources. We do not provide a complete listing of all the functions that should appear in this module, but include some of the more essential and understandable functions.

module *define_interface*

- defines initial settings for service levels, workloads, event rates, and returns a triplet $(\vec{s}, \vec{w}, \vec{r})$

  for example: initial service level setting $\vec{s} = (1, \ldots, 1)$

  initial workloads $\vec{w} = (1, \ldots, 1)$

  initial event rates $\vec{r} = (0, \ldots, 0)$
- provides modules for computing

  runtime $r_{a,h}(w_a, s_a)$ of application a

  latency $\lambda_{a,h}(w_a, s_a)$ of application a

  memory requirement $m_{a,h}(w_a, s_a)$ of application a

  communication time $c_{a,a'}(w_a, s_a)$ for applications $a \prec a'$

  system benefit $\bar{U}(\vec{s}, \vec{w}, \vec{r})$

Figure 5: Module define_interface.

4.2 Module initial_allocation

The module ***initial_allocation*** constructs an allocation of applications to hosts such that their runtimes are minimized. However, runtime minimization cannot be expected to be fully achieved due to limited processor power and memory. It is important to realize that the minimization of runtimes is not the overall objective of an allocation algorithm, but a mechanism for producing a reasonable initial allocation. Figure 6 contains the heuristic used by this module. The allocation is represented as a function $host : A \rightarrow H$, and the heuristic strategy follows a two-dimensional bin-packing approach.

module *initial_allocation*

Input: parameters $\vec{s}, \vec{w}, \vec{r}$
Output: function $host : A \rightarrow H$
implementation
procedure *host;*
initialize *cpu_availabile(h)* := 0.7; *mem_availabile(h)* := *mem(h);*
for each path p do
  for each application a on p do
    assign a to host h such that $\lambda_{a,h}(w_a, s_a)$ is minimum
      subject to $r_{a,h}(w_a, s_a)/\pi_p \leq cpu\_available(h)$
      and $m_{a,h}(w_a, s_a) \leq mem\_available(h)$;
          -- latency $\lambda_{a,h}(w_a, s_a) := r_{a,h}(w_a, s_a)$ + *queuing delay*
    reduce *cpu_available(h)* by $r_{a,h}(w_a, s_a)/\pi_p$;
    reduce *mem_ available(h)* by $m_{a,h}(w_a, s_a)$;

Figure 6: Module initial_allocation.

Module *feasibility_test*

The module ***feasibility_test*** implements a test to analyze the feasibility of a solution. A *feasible* solution is a function *host*: $A \rightarrow H$ that satisfies all the allocation constraints identified in section 3.1. The test requires invoking functions provided by the *define_interface* module and using the returned estimations to determine the feasibility of the allocation. If the feasibility test fails, then the allocation must be modified. If *stopping_criteria2* is not true, then the *optimize_allocation* module is invoked to move applications to different hosts. If *stopping_criteria2* is true, then the parameters ($\vec{s}$, $\vec{w}$, $\vec{r}$) are changed such that the overall utility is decreased. Since we assume montonicity, decreasing these parameters results in lower resource needs. Once a feasible solution is found, the parameters ($\vec{s}$, $\vec{w}$, $\vec{r}$) are changed such that overall utility is increased unless *stopping_criteria1* is true.

At this level we have to deal with resource contention for all resources. In our proposed model we have considered the processor, memory, and network links. For the processor and memory, the feasibility test must determine if the utilization thresholds are not violated. Since contention is present, the latency, defined as the time to complete processing, for a path must be less than the required deadline minus the start time. Contention is encountered in both the processor and the network link. For the processor on a time-shared operating system like UNIX, direct analysis of the response time due to dynamic priority round-robin scheduling can be performed to determine the latency of a single application. For communication delays each pair of dependent applications $a \prec a'$ on different hosts gives rise to a *communication task* $c_{a,a'}$. The *size* of $c_{a,a'}$ is specified as the number of output bits or packets produced by application a. The latency of transmission depends on the technical network properties and the queuing delays due to the current network traffic. The latency of a path is defined as the summation of the latencies of all the applications and communication tasks belonging to the path.

4.3 Module *optimize_allocation*

The module *optimize_allocation* is entered when an allocation has failed to pass the feasibility test and *stopping_criteria2* is not satisfied. This module implements functions for modifying the allocation under the conditions of the given parameter settings $\vec{s}$, $\vec{w}$ and $\vec{r}$. This is done by creating a *neighborhood of allocations*. For defining a neighborhood allocation we provide the operator defined below:

$$move(host, a, h) = host'$$

The operator requires the current solution *host*, an application a, and a target host h as parameters and returns a new solution *host'* that is equal to *host* except for moving application a to host h if possible. The operator results in the assignment of application a to host h. For a given allocation function *host* : $A \rightarrow H$, the neighborhood N(*host*) can be defined as the allocation function.

$$\mathrm{N}(host) = \{\, move(host, a, h) \mid a \in A, h \in H \,\}.$$

Other neighborhoods might be necessary to further improve the efficiency and performance of the optimization technique. The neighborhood functions are the basis for heuristic optimization algorithms to improve the allocation for given parameter settings. General purpose local search optimization heuristics such as *simulated annealing*, *tabu search*, and *evolutionary algorithms* can be implemented as swappable components within this module.

Module *modify_parameters*

The module *modify_parameters* is responsible for modifying the performance parameters $\vec{s}$, $\vec{w}$ and $\vec{r}$. When notified that a feasible solution exists for the current parameter settings $(\vec{s},\vec{w},\vec{r})$, this module will find new parameter settings $(\vec{s}',\vec{w}',\vec{r}')$ that results in a higher system utility. We write this condition as:

$$\bar{U}(\vec{s}',\vec{w}',\vec{r}') > \bar{U}(\vec{s},\vec{w},\vec{r})$$

Due to the monotonicity assumption made in section 3.2, such a selection of parameters results in higher resource requirements. The algorithm must attempt to find a new allocation subject to new parameter settings.

The module *modify_paramters* may also be notified when a feasible solution can not be found for the current parameter settings $(\vec{s},\vec{w},\vec{r})$. The module proceeds to find new parameter settings $(\vec{s}',\vec{w}',\vec{r}')$ that results in a lower system utility. We write this condition as:

$$\bar{U}(\vec{s}',\vec{w}',\vec{r}') < \bar{U}(\vec{s},\vec{w},\vec{r})$$

This selection of parameters results in lower resource requirements due to the monotonicity assumption. This implies that a feasible solution may exist for the new parameters.

5 Related Research

The framework we have presented has been influenced by many growing fields of research in the real-time community. In particular, we have designed our model to allow dynamic resource allocations, permit dynamic profiling, incorporate utility models, and utilize application service levels. In this section, we discuss some of the research that most influenced our model.

DQM [2] uses QoS levels (service levels in our model) to adapt multimedia applications to overload situations. The use of QoS levels enables DQM to gracefully degrade to overload situations. However, DQM uses a worse case execution time, as in [Liu, WCET], to determine application resource usage. It does not reallocate tasks at run-time, only considers one resource, and does not guarantee the optimal, or even near-optimal, set of choices have been made for every situation.

Q-RAM [14] uses a utility function approach to dynamically determine what service levels to choose for a group of applications. Utility can be nearly optimized at run-time by dynamically allocating multiple finite resources to satisfy multiple service levels. A drawback to the model is the use of profiles determined *a priori* that are associated with each service level. In [5], a similar problem is addressed, but the notion of utility is simpler. The same drawback is present in [5] as in [14].

In QuO [18], applications adjust their own service levels to improve performance and adjust to their environment. The model has many drawbacks for dynamic environments. It does not treat all resources within the system as a single set of resources, so reallocations do not occur.

Applications react to the environment on their accord, so there is no way to optimize the set of choices made for all applications.

Burns et al [3] present an explanation on the need for utility-based scheduling in dynamic, real-time environments. Their model includes a set of different service levels, alternatives, for tasks. They also present a manner for elicitation of utility preferences. However, they characterize resource usage on worst case execution time and they do not take many dynamic measures into account, such as workload and event arrival rate.

In [7, 8, 9], Kalegoraki et al use dynamic object profiling techniques to determine resource usage, and resource reallocation techniques are implemented as cooling and heating algorithms to ensure load balancing. A utility function is used to determine what applications to replicate for fault tolerance. Application relations are defined by a graph and referred to as a task. The approach does not include much in the way of a utility optimizationfor resource allocations, except for fault tolerance, and does not include service levels of any type.

In other works, we were mostly concerned with notions of service levels. Liu et al[17], use a notion of service levels where tasks are defined by a mandatory task and an optional task. The optional task's operation may be cut off at anytime to get an output and save resources for other tasks. The optional task's utility increases with time, until it reaches a maximum. In the Elastic Scheduling technique [4], applications are modeled as springs with associated elastic coefficients. The service level for an application is lowered by compressing the application, and the service level is raised by allowing the application to expand. The FLEX language [10] allows programmers to define performance polymorphism to allow a set of alternate algorithms to be executed for one function.

6 Conclusions

In this paper, we have presented a model that characterizes distributed real-time systems operating in dynamic environments. Distributed resources are treated as a pool of resources to be used by the real-time system as a whole. Dynamic environment characteristics are modeled by event arrival rates, workloads, and service levels. The notions of utility and service levels provide a means for graceful degradation and give a manner to optimize the allocation of resources. A framework is presented to produce feasible, optimal allocations even when applications receive unknown event arrival rates and process dynamic amounts of workload.

Future work includes producing a more practical service level parameter definition, integrating fault tolerance into the utility functions as done in [9], and allowing for load sharing techniques among replicas.

References

[1] Ballou, K. and Miller, J., "On-board Cloud Contamination Detection with Atmospheric Correction," *NASA's Earth Science Technology Conference 2002*, Pasadena, CA, June 2002.

[2] Brandt, S., and Nutt G., "Flexible Soft Real-Time Processing in Middleware," *Real-Time Systems*, pp. 77-118, 2002.

[3] Burns, A., et al, "The Meaning and Role of Value in Scheduling Flexible Real-Time Systems," *Journal of Systems Architecture*, vol. 46, pp. 305-325, 2000.

[4] Buttazzo, G., et al., "Elastic Scheduling for Flexible Workload Management", *IEEE Transactions on Computers,* Vol. 51, No. 3, pp. 289-302, March 2002.

[5] Chen, L., et al, "Building an Adaptive Multimedia System Using the Utility Model" *International Workshop on Parallel & Distributed Real-Time Systems*, San Juan, Puerto Rico, pp 289-298, April 1999.

[6] Jain, S., et al, "Collaborative Problem Solving Agent for On-Board Real-time Systems," *Proceedings of 16th International Parallel and Distributed Processing Symposium*, pp.15-19, Ft. Lauderdale, FA, April 2002.

[7] Kalogeraki, V., Melliar-Smith, P., and Moser, L., "Dynamic Scheduling for Soft Real-Time Distributed Object Systems", *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pp. 114-121, Newport Beach, California, 2000.

[8] V. Kalogeraki, P. M. Melliar-Smith and L. E. Moser, "Using Multiple Feedback Loops for Object Profiling, Scheduling and Migration in Soft Real-Time Distributed Object Systems," *In the Proceedings of the Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Saint Malo, France, May 1999.

[9] Kalogeraki, V., Moser, L. and Melliar-Smith, P. "Dynamic Modeling of Replicated Objects for Dependable Soft Real-Time Distributed Object Systems", *Fourth IEEE International Workshop on Object-Oriented Real-time Dependable Systems*, Santa Barbara, CA, 1999.

[10] Kenny, K. and Lin, K., "Building Flexible Real-Time Systems using the FLEX Language", *IEEE Computer*, Vol.24, No.5, pp.70-78, May 1991.

[11] Klein, M., et al., "A Practitioner's Handbook for Real-Time Analysis" Kluwer, 1993.

[12] Krishnamurthy, et al, "Integration of QoS-Enabled Distributed Object Computing Middleware for Developing Next-Generation Distributed Applications," *Proceedings of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems*, Snowbird, Utah, June 2001.

[13] Kuo, T. and Mok, A., "Incremental reconfiguration and load adjustment in adaptive real-time systems," *IEEE Transactions on Computers,* 46(12), pp. 1313-1324, December 1997.

[14] Lee, C., et al, "A Scalable Solution to the Multi-Resource QoS Problem," *Proceedings of the 20th IEEE Real-Time Systems Symposium*, December 1999.

[15] Lehoczky, J., "Real-time queueing theory," *Proceedings of the 17th IEEE Real-Time Systems Symposium,* pp. 186-195, IEEE Computer Society Press, 1996.

[16] Liu, C. and Layland, J., "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM,* 20, pp. 46-61, 1973.

[17] Liu, J., et al, "Algorithms for Scheduling Imprecise Computations," *IEEE Computer*, vol. 24 (5), pp. 58-68, May 1991.

[18] Loyall, J., et al, "Emerging Patterns in Adaptive, Distributed Real-Time Embedded Middleware," *9th Conference on Pattern Language of Programs*, Monticello, Illinois, Sept. 2002.

[19] Sha, L., Klein, M., and Goodenough, J., "Rate monotonic analysis for real-time systems," in *Scheduling and Resource Management,* Kluwer, pp. 129-156, eA. M. van Tilborg and G. M. Koob, 1999.

[20] Stewart, D. and Khosla, P., "Mechanisms for detecting and handling timing errors," *CACM*, 40(1), pp. 87-93, Jan. 1997.

[21] Tia, T., et al., "Probabilistic performance guarantee for real-time tasks with varying computation times," *Proceedings of the 1st IEEE Real-Time Technology and Applications Symposium,* pp. 164-173, IEEE Computer Society Press, 1995.

[22] Verhoosel, J., et al., "A Model for Scheduling of Object-Based, Distributed Real-Time Systems," *Journal of Real-Time Systems*, 8(1), pp. 5-34, Kluwer Academic Publishers, January 1995.

[23] Welch, L., Stoyenko, A., and Marlowe, T., "Modeling Resource Contention Among Distributed Periodic Processes Specified in CaRT-Spec," *Control Engineering Practice*, 3(5), pp. 651-664, May 1995.

[24] Welch, L., et al, "Adaptive QoS and Resource Management Using A Posteriori Workload Characterizations," *The IEEE Real-Time Technology and Applications Symposium,* pp. 266-275, June 1999.

[25] Welch, L., et al, "Specification and Modeling Of Dynamic, Distributed Real-Time Systems," *The IEEE Real-Time Systems Symposium*, IEEE Computer Society Press, pp. 72-81, December 1998.

[26] Welch, L. and Shirazi, B., "A Dynamic Real-Time Benchmark for Assessment of QoS and Resource Management Technology," *The IEEE Real-Time Technology and Applications Symposium*, pp. 36-45, June 1999.

[27] Welch, L., Pfarr, B. and Tjaden, B., "Adaptive Resource Management Technology for Satellite Constellations," *The Second Earth Science Technology Conference (ESTC-2002)*, Pasadena, CA, June 2002.

[28] Welch, L., et al., "Adaptive Resource Management for On-board Image Processing Systems," *Journal. of Parallel & Distributed Computing Practices-* issue on parallel & distributed real-time systems, Nova Science Publishers (to appear).